Part 3A
For NULL-pointer dereference, I basically mimicked what is done when creating the user stack by adding an inaccessible page starting at virtual address 0. To do this, in exec before allocating pages for the user code, I allocate an empty page that is mapped to virtual address 0x0 (using allocuvm) and then clear its user bit (PTE_U) to make it inaccessible to users. So if the user program tries to access any memory in this page, when the hardware does the translation, it will see that the PTE's user bit is cleared and will end up throwing a pagefault error.

To implement the memory protection functionality, I added the system calls mprotect and munprotect. In these functions, I leveraged the code in xv6 to first get the arguments passed into the syscall (using argint and argptr). Argptr handles most of the error handling to check if the address is valid. The only thing I had to check myself was if it was page aligned. Once I got the arguments and error checked them, I used myproc to get the pgdir of the current process and then walkpgdir to get the page table entry of the specified address range. For each page table entry, I either clear the PTE_W bit (mprotect) or set it (munprotect). To make sure that the hardware knows of this change, I write to the cr3.

Part3B
To create a kernel thread, I copied fork() and instead of allocating a new address space, I copied the pointer of the parent process to the new thread. Additionally, to ensure that when the thread runs it runs fcn and has access to the arguments, I set the trap frame instruction pointer (eip) to fcn and stack pointer (esp) to the top of the stack that was passed in and also copied the arguments and return address to the top. So esp ended up pointing to the return address. Join() is the same as wait() except for now I check that the parent and child process have the same pgdir to determine if it's a thread. I also no longer free up the address space. To keep track of the thread stack, I added an element to the proc struct and store the stack pointer there. I also added a thread count to the proc struct. It is initialized to 0 in allocproc, incremented by 1 in clone, and decremented by 1 in join. This is then used in wait to make sure that there are no more active threads for a process so that the address space can be freed.

For locks, I leveraged the code from wiki to add a fetch-and-add atomic operation in x86.h, which I use to implement the ticket spin lock from OSTEP.

For the thread user API, in thread_create I call malloc() to allocate a page of memory for the thread stack and then pass that in along with the user arguments to clone(). In thread_join, I call join() and if it returns a valid PID, I free up the stack pointer that it places in the argument that is passed in to it.

When a process' address space is grown, I added a lock acquire in growproc() to make sure the address space of a process and its children threads is synchronized. Other than a lock, I also added a loop through the process table to find all of the process' child threads and update their address space size (p->sz).